의존성과 계층화

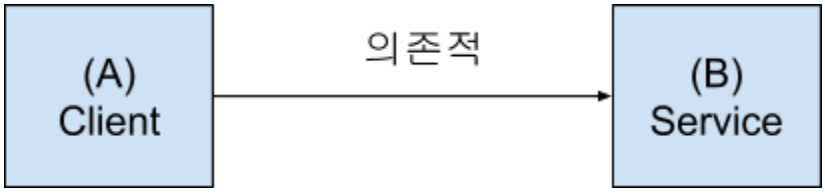
모든 소프트웨어는 의존성(dependency)을 가지고 있다.

1. 같은 기반 코드를 이용하는 코드에 대한 first-party 의존성
2. 외부 어셈블리에 대한 third-party 의존성
3. MS .Net 프레임워크에 대한 보편적인 의존성

변화를 수용할 수 있는 코드를 작성하기 위해서는 의존성을 효과적으로 관리해야 한다!

의존성? (dependency)

별개의 두 엔티티 사이의 연관관계로 인해 어느 한 엔티티가 다른 엔티티의 기능 없이는(존재 없이는) 자신의 기능을 실행하지 못하는 관계를 의미



A가 B에 의존적이다. 하지만, B는 A에 의존적이지 않다.
관계 서술 방법 : A가 B의 클라이언트이고, B는 A의 서비스이다.

예제1)

1. Console Application 생성 (SimpleDependency.exe)
2. Class library 생성 (MessagePrinter.dll)
3. SimpleDependency 어플리케이션에 MessagePrinter 라이브러리 참조추가
4. 빌드 하면, 바이너리 출력 폴더에 참조 라이브러리까지 복사된다

→ 하지만, 실행하면 런타임에 실제 MessagePrinter.dll를 로딩하지 않는다. (모듈 창으로 확인)
→ using MessagePrinter; 를 추가하여도 로딩하지 않는다. (using 구문은 CLR이 실행할 명령을 생성하지는 않는다)

예제2)

1. 콘솔 어플리케이션에 아래 구문 구현

```
var service = new MessagePrintingService();
service.PrintMessage();
Console.ReadKey();
```

→ 의존하는 모듈을 사용하였으므로 런타임 시 로딩하지 않으면, MessagePrintingServer 인스턴스를 생성할 수 있는 방법이 없다.

이러한 의존성을 first-party 의존성이라고 한다.

콘솔 어플리케이션과 클래스 라이브러리 모두 사용자가 생성한 프로젝트이므로 언제든지 접근하여 다시 빌드할 수 있기 때문에 의존성 모듈에 항상 액세스가 가능하다.

하지만~!
또한, 이 두 프로젝트는 닷넷 프레임워크 어셈블리에 의존성을 가지고 있다!

기본 참조 목록

Windows Form 어플리케이션 → System.Windows.Form 어셈블리 참조
WPF 어플리케이션 → WindowsBase, PresentationCore, PresentationFramework 어셈블리 참조

모든 비주얼 스튜디오 프로젝트 템플릿
→ 비주얼 스튜디오가 설치된 디렉토리 아래의 /Common7/IDE/ProjectTemplates/ 에 언어별로 구분되어 저장되어 있다. → 실제 프로젝트 인스턴스를 생성하는 데 필요한 약간의 로직이 포함되어 있다.

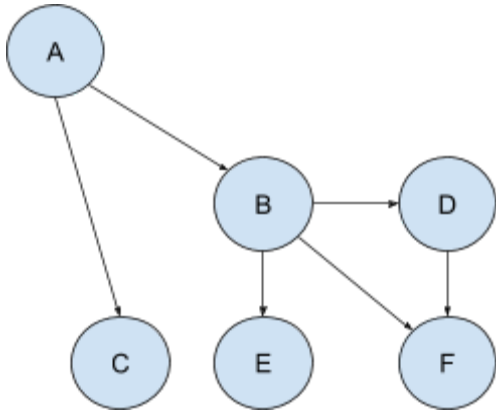
Third-Party 의존성

서드파티 의존성을 선택하는 가장 중요한 이유는 어떤 기능이나 인프라스트런처를 직접 구현하느라 많은 노력을 투자하는 대신 이미 만들어진 것을 가져와 적절히 활용할 수 있기 때문

1. 서드파티 라이브러리는 비주얼 스튜디오 솔루션의 Dependencies 폴더에 모아두자!
2. NuGet을 이용하여 의존성을 관리하자!

유향 그래프를 이용한 의존성 모델링

의존성 다이그래프 (dependency digraph)
→ 엔티티를 노드라고 생각하고 종속적인 코드에서 의존성을 제공하는 코드로 방향이 있는 엣지를 그려서 표현



화살표가 시작되는 노드는 종속적인 컴포넌트이며, 화살표가 가리키는 노드는 의존성을 제공하는 컴포넌트이다.

그러나, 상속(inheritance), 결합(aggregation), 조합(composition), 연관(association)의 의존성을 관리해야 하는지는 당장 알지 못한다.

순환 의존성

그래프 이론에서 발견할 수 있는 다른 사항은 방향을 가진 그래프는 순환되는 형태가 될 수도 있다는 점이다.

하지만, 어셈블리의 경우, 순환 의존성이 불가능하다. → 비주얼 스튜디오 오류 메시지 출력
어셈블리간의 순환 의존성은 완벽히 차단해야 하며 반드시 피해야 한다!

의존성 관리하기

패턴(pattern) → 클래스와 인터페이스 사이에서 반복적으로 발생하게 되는 협업 등을 규정하여 집대성한 것

안티패턴(anti-pattern) → 코드의 유연성을 해치기 때문에 사용을 지양해야 할 패턴

구현과 인터페이스의 비교

컴파일 시에는 인터페이스의 클라이언트는 해당 인터페이스에 대한 어떤 구현체가 사용되고 있는지에 대해 전혀 알 필요가 없다. 알게 된다면, 오히려 잘못된 가정으로 인해 의존성이 더 높아지게 된다.

new 키워드의 코드 스멜

인터페이스 → 어떤 일을 수행할 수 있는지를 서술
클래스 → 어떻게 특정 작업을 수행할 것인지 서술, 실제 구현에 대한 상세 내용을 알고 있음

코드스멜(code smell) → 어떤 코드가 잠재적으로 문제가 있을 수 있음을 표현하는 단어

코드스멜은 반드시 해결해야 할 부채가 있으며, 그 기술 부채가 해결되지 않고 계속해서 남아있게 되면 앞으로 더욱 고치기 어려워질 것임을 암시적으로 나타낸다.

new키워드의 사용은 부적절한 결합의 한 예이다. (객체 인스턴스의 직접적인 생성)

(new 사용으로 인한 코드의 적응성이 저하되는 예제)

```
public class AccountController
{
    private readonly SecurityService securityService;

    public AccountController()
    {
            this.securityService = new SecurityService();
    }

   [HttpPost]
   public void ChangePassword(Guid userID, string newPassword)
   {
            var userRepository = new UserRepository();
            var user = userREpository.GetByID(userID);
            this.securityService.ChangeUsersPassword(user, newPassword);
   }
}
```

- AccountController 클래스는 SecurityService 클래스와 UserRepository 클래스의 구현에 대해 영원히 의존적이다.
- AccountController는 단위 테스트를 하기가 어려워짐
- SecurityService.ChangeUserPasword 메소드는 클라이언트가 User객체를 로드할 수 밖에 없도록 만듬

(SecurityService클래스의 클라이언트를 향상시키기 위한 코드 예제)

```
...
  [HttpPost]
  public void ChangePassword(Guid userID, string newPassword)
  {
          this.securityService.ChangeUsersPassword(userID, newPassword);
  }

  public void ChangeUserPassword(Guid userID, string newPassword)
  {
          var userRepository = new UserRepository();
          var user = userRepository.GetByID(userID);
          user.ChangePasword(newPassword);
  }
```

- AccountController 클래스는 개선되지만, ChangeUserPassword 메서드는 UserRepository 객체의 인스턴스를 직접 생성하는 문제점은 여전히 가지고 있음

객체 생성에 대한 대안

인터페이스를 기초로 한 코딩

Security 클래스의 실제 구현을 인터페이스 뒤로 숨기는 것. 인터페이스에만 의존하게 만든다.

(ISecurityService 인터페이스를 참조하도록 변경한 예제)

```
public class AccountController
{
   private readonly ISecurityService securityService;

   public AccountController()
   {
          this.securityService = new SecurityService();
   }

  [HttpPost]
  public void ChangePassword(Guid guid, string newPassword)
  {
          securityService.ChangeUsersPassword(user, newPassword);
  }
}
```

```
public interface ISecurityService
{
    void ChangeUsersPassword(Guid userID, string newPassword);
}

public class SecurityService : ISecurityService
{
    Public ChangeUserPassword(Guid userID, string newPassword)
    {
        ...
    }
}
```

- 인터페이스를 참조하도록 수정했지만, 완변하지 않음
- 여전히 SecurityService 클래스의 생성자를 호출함으로 의존성을 가지고 있음
- 완전히 분리하기 위해 의존성 주입 기법 활용

의존성 주입 (dependency injection) 기법 활용하기

(의존성 주입 기법을 이용하여 수정한 예제)

```
public class AccountController
{
   private readonly ISecurityService securityService;

   public AccountController(ISecurityService securityService)
   {
        If (securityService == null)
           throw new ArgumentNullException("securityService");
        this.securityService = securityService;
   }

  [HttpPost]
  public void ChangePassword(Guid guid, string newPassword)
  {
        securityService.ChangeUsersPassword(user, newPassword);
  }
}
```

- SecurityService를 직접 생성하는 대신, 다른 클래스에게 ISecurityService 인터페이스를 구현한 객체를 제공해 줄 것을 요구
- 의존성이 완전히 제거됨

추종자 안티패턴 (Entourage anti-pattern)

단순한 것을 요구했는데 관련된 모든 것들이 뒤따라오는 상황

인터페이스와 그에 관련된 의존성들은 동일한 어셈블리에 존재해서는 안된다!

(위의 예제의 잘못된 예)
인터페이스의 구현이 인터페이스 자체와 동일한 어셈블리에 존재하고 있다. → Services 프로젝트 내에 인터페이스 코드와 구현 코드가 같이 있음
→ Controllers 프로젝트만 따로 빌드하더라도 bin 디렉토리에서 NHibernate 관련 어셈블리들을 발견하게 됨 → 여전히 묵시적 의존성이 존재한다. → 각각의 어셈블리와 느슨하게 결합된 (loosely coupled) 형태는 구현하지 못함

구현체를 인터페이스와 분리하여 별도의 어셈블리에 구현하는 것은 일반적인 규칙이다
→ 계단 패턴(Stairway pattern)을 적용!

계단 패턴

클래스와 인터페이스를 관리하는 올바른 방법
인터페이스와 실제 구현 클래스를 서로 다른 어셈블리에 정의함으로써 두 어셈블리를 독립적으로 관리할 수 있고, 클라이언트는 인터페이스 어셈블리 하나만 참조하게 구현

원칙 : 인터페이스는 어떠한 외부 의존성도 가져서는 안된다.

서드파티 라이브러리 문제 → 계단 패턴이 아닌 추종자 안티패턴을 바탕으로 패키징된다.
→ 회피하기 위해 서드파티에 대한 의존성을 퍼스트파티 의존성을 이용해 숨기는 간단한 인터페이스를 정의한다. → 하지만, 의존성을 바꾸는 인스턴스 작업 양도 많을 뿐만 아니라 의존성을 지속적으로 유지 및 관리해야 하는 불편을 감수해야 한다.

의존성 해석하기

퓨전로그 (fusion log)
: CLR이 런타임 시점에 실패했던 어셈블리 바인딩을 디버깅하기 위한 유용한 도구.

활성화 방법 → 아래 레지스트리를 수정 혹은 추가
HKLM\Software\MIcrosoft\Fusion\ForceLog 1

HKLM\Software\MIcrosoft\Fusion\LogPath C:\FusionLogs
(로그경로는 사용자가 원하는 폴더로 변경가능)

퓨전로그 관리 UI프로그램
→ C:\Program Files(x86)\Microsoft SDKs\Windows\v7.0A\Bin\x64\FUSLOGVW.exe

서비스
: 어셈블리에 비해 클라이언트와 호스트 서비스의 결합은 비교적 느슨한 결합이며 조금 더 나은 결합이기는 하지만, 이 역시 비용이 수반된다.

RESTful 서비스
: 클라이언트의 의존성을 최소화할 수 있음

계층화

소프트웨어 컴포넌트를 각 기능의 수평적 계층으로 취급하고 이들을 바탕으로 전체 어플리케이션을 구성해 나가도록 유도하는 아키텍처 패턴

의존성의 방향은 항상 아래를 향하게 되며, 최하위 계층은 어떤 의존성도 갖지 않음 (엄밀히 말하면, 서드파트 의존성은 가질 수 있다)

레이어(Layer)
: 논리적으로 구분, 개념적으로 구분

티어(Tier)
: 물리적으로 구분. 개수는 어플리케이션이 배포되는 머신의 개수와 관련. 어플리케이션을 여러 티어로 나누어 배포하게 되면 확장성 면에서 큰 장점을 제공

수직적확장
: 메모리나 프로세스 등을 추가하여 컴퓨터의 성능을 끌어올리는 방법. 이를 통해 하나의 머신이 보다 많은 작업을 처리

수평적확장
: 여러 개의 머신을 배치하고 로드 밸런스를 이용해 사용자의 요청을 여력이 있는 머신으로 재분배하여 더 많은 처리

두 개의 계층

사용자 인터페이스와 데이터 액세스에 각각 집중하는 2개의 논리적인 어셈블리 집합을 구성

사용자 인터페이스

- 어플리케이션을 사용하기 위한 방법 제공
- 사용자에게 데이터와 정보를 표시
- 사용자의 요청을 질의나 명령 형태로 받아들임
- 사용자가 입력한 정보에 대한 유효성 검사
- 데이터 액세스 계층 위에 존재 (의존)
- 데이터 액세스 계층을 구현한 어셈블리를 직접 참조하지 않고 데이터 액세스 인터페이스를 참조한다.

데이터 액세스

- 데이터에 대한 질의를 제공
- 객체 모델과 관계형 모델 간의 직렬화/역직렬화를 수행
- 여러 기술에 대한 의존성을 전혀 갖지 않는 인터페이스에 의해 최대한 가려져야 한다.
- 모든 인터페이스는 서드파티에 대한 의존성을 갖지 않아야만 클라이언트가 선택한 그 어떤 기술과도 분리하여 관리할 수 있다.

장점

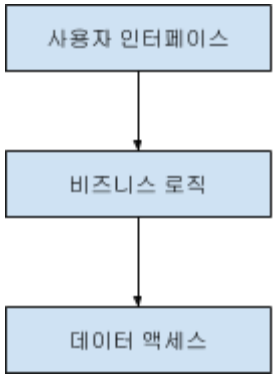
- 약간의 유효성 검사를 제외하고는 어플리케이션 로직이 거의 또는 아예 존재하지 않는 경우 유용
- 주로 데이터에 대한 CRUD작업만을 수행하는 경우
- 개념 증명을 통한 실현 가능성을 빠르게 확인 가능 (기간이 짧은 경우)

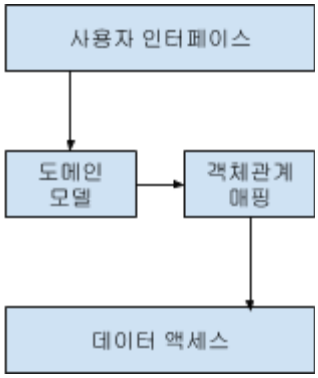
단점

- 방대한 로직을 가지고 있거나 로직의 주제가 변경되는 경우, 유연성과 유지보수성을 저하
- 아키텍처가 단 몇 주만에 바뀌게 되는 경우, 빠른 피드백을 얻기 위해 감수했던 부분들은 그 가치를 상실

세 개의 계층

어플리케이션에 로직 계층을 추가하여 보다 복잡한 처리를 캡슐화





비즈니스 로직

- 사용자 인터페이스 계층으로부터 명령을 처리
- 비즈니스 도메인을 모델링하여 비즈니스 프로세스, 규칙, 업무, 흐름 등을 정의
- 도메인 모델은 하위 계층은 물론 특정 구현 기술에 대한 의존성이 반드시 제거 되어야 함

횡단 관심사 (cross-cutting concerns)

간혹 컴포넌트 역할의 하나의 계층으로 제한하기 쉽지 않은 경우 존재 → 어플리케이션 전체에 적용되어야 하는 컴포넌트 존재
→ 횡단 관심사의 요건들을 정의 → 캡슐화된 기능으로 떼어 내어 코드에 적용하면 코드에 대한 간섭을 최소화 할 수 있음 → 관점지향 프로그래밍 기법

관점지향 프로그래밍 (AOP, Aspect-Oriented Programming)
: 횡단 관심사(관점)을 코드 내의 여러 계층에 적용하는 방법. NuGet에서 AOP로 검색

비대칭 계층화

실용성에 대한 추구와 계층화가 경우에 따라 일부 요청에 부하를 더하거나 혹은 비효율적으로 처리하게 되는 경우에 대한 고려
→ 명령/질의 책임 격리 (CQRS, Command/Query Responsibility Segregation) 패턴

명령/질의 분리 (CQS)

명령

- 어떤동작을 위해서 의무적으로 호출해야 하는 것, 코드로 하여금 필요한 작업을 실행
- 시스템의 상태를 변경할 수는 있지만, 값을 리턴해서는 안됨
- 호출 순서에 조금 더 주의를 기울여야 함 (객체 상태를 변경시키므로…)

질의

- 데이터에 대한 요청이며, 코드로 하여금 필요한 데이터를 가져오게 한다.
- 자신을 호출한 클라이언트에게 데이터를 리턴, 시스템 상태는 변경안함
- 객체의 상태에 아무런 영향을 미치지 않으므로 호출 순서를 마음대로 할 수 있다.

(명령 : CQS를 준수하는 메서드와 그렇지 않은 메서드)

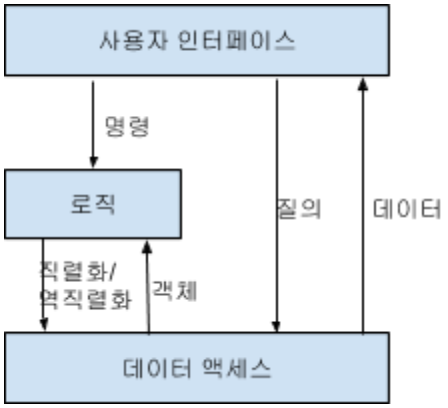
```
// CQS를 준수하는 메서드
public void SaveUser(string name)
{
    session.Save(new User(name));
}
// CQS를 준수하지 않는 메서드
public User SaveUser(string name)
{
    var user = new User(name);
    session.Save(user);
    return user;
}
```

(질의 : CQS를 준수하는 메서드와 그렇지 않은 메서드)

```
// CQS를 준수하는 메서드
public IEnumerable<User> FindUserByID(Guid userID)
{
    return session.Get<User>(userID);
}
// CQS를 준수하지 않는 메서드
public IEnumerable<User> FindUserByID(Guid userID)
{
    var user = session.Get<User>(userID);
    user.LastAccessed = DateTime.Now;
    return user;
}
```

명령/질의 책임 격리 (CQRS)

도메인 모델을 바탕으로 한 3계층 아키텍처를 구현할 때 적용 가능
→ 도메인 모델은 애플리케이션의 명령 측 영역에서만 사용
→ 질의 측 영역에는 보다 단순한 2계층 모델이 사용



요약

1. 장기적으로 건실하고, 적응성이 뛰어나며, 변화를 수용할 수 있도록 프로젝트를 유지하는 것은 의존성 관리 방법에 좌우된다.
2. 의존성은 개별 클래스와 메서드를 거쳐 어셈블리 참조부터 고수준의 컴포넌트 아키텍처에 이르기까지 모든 수준에서 반드시 관리되어야 한다.